Sidrah Hashmi - 100915053
Nayab Khan - 100922060
Zainab Syed - 100924129

Implementation Part:

```
# Final Project: Real-Time Text Autocompletion System
# Course: INFR-2820U (Data Structures and Analysis of Algorithms)

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class StandardTrie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):  # method to insert a word into the trie
        word = word.lower()  # Convert to lowercase
        node = self.root
        for char in word:    # if character does not exist we create a new node
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True     # mark the end of the word

    def search(self, prefix):    #method to search for a prefix in the trie
        prefix = prefix.lower()  # Convert to lowercase
        node = self.root
        for char in prefix:
            if char not in node.children:
                # If no match, return all words
                return self._get_words_from_node(self.root, "")
            node = node.children[char]
        return self._get_words_from_node(node, prefix)

    def _get_words_from_node(self, node, prefix):   # recursive method to all collect words
        words = []
        if node.is_end_of_word:
            words.append(prefix)    #if its a word, add to results
        for char, child in node.children.items():    #recursively get words from each child node
            words.extend(self._get_words_from_node(child, prefix + char))
        return words
```

```
class TernaryTrieNode:
    def __init__(self, char):
        self.char = char
        self.left = self.middle = self.right = None
        self.is_end_of_word = False

class TernaryTrie:
    def __init__(self):
        self.root = None

    def insert(self, word):
        word = word.lower()  # Convert to lowercase
        self.root = self._insert(self.root, word, 0)

    def _insert(self, node, word, index):   #helper recursive insert method
        char = word[index]
        if node is None:
            node = TernaryTrieNode(char)  #create a node if it doesn't exist.
        if char < node.char:
            node.left = self._insert(node.left, word, index)
        elif char > node.char:
            node.right = self._insert(node.right, word, index)
        else:
            if index + 1 == len(word):
                node.is_end_of_word = True
            else:
                node.middle = self._insert(node.middle, word, index + 1)  #go middle
        return node

    def search(self, prefix):   # search for a prefix in a ternary trie
        prefix = prefix.lower()  # Convert to lowercase
        node = self._search(self.root, prefix, 0)
        if not node:
            # If no match, return all words
            return self._get_words_from_node(self.root, "") #return all
        return self._get_words_from_node(node.middle, prefix)

    def _search(self, node, prefix, index):   # recursive helper to find prefix node
        if not node:
            return None
        char = prefix[index]
        if char < node.char:
            return self._search(node.left, prefix, index)
        elif char > node.char:
            return self._search(node.right, prefix, index)
        elif index + 1 == len(prefix):
            return node  # prefix match found
        else:
            return self._search(node.middle, prefix, index + 1)

    def _get_words_from_node(self, node, prefix):  # collect all words starting from a node
        words = []
        if not node:
            return words
        if node.is_end_of_word:
            words.append(prefix)   #add word if node marks its end
        words.extend(self._get_words_from_node(node.left, prefix))
        words.extend(self._get_words_from_node(node.middle, prefix + node.char))
        words.extend(self._get_words_from_node(node.right, prefix))
        return words  #above, we are recursively collecting words in left, middle, right subtrees

def main():
    # New set of words
    words = ["apple", "apricot", "banana", "blueberry", "blackberry", "cherry", "date", "elderberry",
"fig", "grape"]
    #initialize both trie structures
    standard_trie = StandardTrie()
    ternary_trie = TernaryTrie()
    #insert all words into both tries
    for word in words:
        standard_trie.insert(word)
        ternary_trie.insert(word)

    while True:  #interactive loop for user input
        prefix = input("Enter a prefix (or type 'exit' to quit): ").strip()
        if prefix.lower() == 'exit':
            break
        #show suggestions from both tries
        print("\nStandard Trie Suggestions:", standard_trie.search(prefix))
        print("Ternary Trie Suggestions:", ternary_trie.search(prefix))
        print("-" * 50)

if __name__ == "__main__":
    main()
```

Analysis Part:

Part a: Space Efficiency Analysis

When analyzing space efficiency between the Standard Trie and Ternary Trie, it's important to focus on how the data structures scale with the number of words (N) and their average length (L). In our implementation, the Standard Trie uses a “TrieNode” class where each node maintains a dictionary for its children. This results in a large number of pointers, potentially up to 26 per node for each lowercase letter. As we insert more words, especially ones with overlapping prefixes, this structure continues to consume memory even when not all slots are used. The worst-case space complexity for this implementation is O(N × L), where each character in every word leads to a new node. This redundancy becomes more noticeable when multiple words share common prefixes, leading to unnecessary replication of similar structures in memory.

In contrast, the Ternary Trie, represented by our “TernaryTrieNode” class, maintains only three pointers per node (left, middle, and right). This more compact representation ensures that we only maintain a minimal branching structure and avoid the overhead of storing an entire alphabet's worth of links at each level. Because the Ternary Trie builds its structure using binary search logic over characters, its space complexity in practical terms is closer to O(N). This design efficiently reuses prefixes and only creates new branches when necessary, making it significantly more space-efficient than the Standard Trie, particularly for larger datasets.

Part b: Time Efficiency Analysis

Time complexity determines how efficiently we can insert and retrieve words or prefixes in our autocomplete system. For the Standard Trie, both insertion and search operations follow a linear pattern, where each character is processed in sequence. This gives a time complexity of O(L) for insertion and O(L) for search, where L is the length of the word or prefix being processed. For autocomplete functionality, once a prefix is found, the program collects all valid completions from that point onward, adding an extra O(W) where W is the number of matching words. Therefore, the full autocomplete operation results in O(L + W) complexity.

The Standard Trie performs best with smaller datasets due to its direct access nature via dictionaries, which allows each character to be reached in constant time. In contrast, the Ternary Trie follows a binary search approach. In our implementation, each node comparison guides traversal toward the correct character by checking left, middle, or right pointers, similar to how binary search works.

As a result, the Ternary Trie achieves an average-case complexity of O(log N + L) for insertion and search. Here, log N accounts for the character comparisons across branches, while L represents the characters in the word. Autocomplete in the Ternary Trie remains efficient with complexity O(log N + W), where W is again the number of completions found. This makes the Ternary Trie ideal for large-scale datasets where the reduced depth of traversal significantly improves search speed. Overall, while the Standard Trie offers straightforward speed for shorter or fewer inputs, the Ternary Trie scales better and performs faster with more complex or voluminous word sets due to its balanced character comparisons and structured pathfinding.

Part c: Impact of Ranking Algorithm on Complexity

Introducing a ranking algorithm into our autocomplete system adds an intelligent layer to user experience by prioritizing frequently searched words. The algorithm works by assigning a frequency counter to each word, keeping track of how many times each word is selected after a query. This allows the system to learn user behavior over time. For example, if the user frequently searches for the word "Sample" after typing the prefix "Sam," and less frequently selects alternatives like "Same" or "Samplers," the system will recognize "Sample" as the most relevant and list it at the top of the suggestions. This ranking logic provides a more customized and user-friendly output.

Implementing this ranking mechanism introduces a slight increase in space complexity due to the need to store and update frequency counters for each word, resulting in an additional O(N) space requirement. However, this is relatively minor and doesn't significantly affect the overall memory use of the data structure. When it comes to time complexity, ranking impacts the post-processing phase after the prefix has been matched and words have been retrieved. Specifically, sorting the results based on frequency introduces a time complexity of O(W log W), where W is the number of matched words. Importantly, this sorting step is separate from the actual search or insertion process. The base search complexities of the underlying data structures remain unchanged—O(L) for the Standard Trie and O(log N + L) for the Ternary Trie. Autocomplete retrieval is slightly slower due to the ranking sort step, but this does not change the worst-case performance of the trie itself.

The ranking algorithm brings a small and controlled increase in space and time complexity. Its space impact is minor and its time impact is isolated to result sorting, without altering the core search operations. The trade-off is highly beneficial, as the ranking enables more intelligent predictions and improves the overall relevance of suggestions. This optimization maintains overall efficiency while enhancing user interaction. As a result, both the Standard and Ternary Tries continue to perform effectively, with the Ternary Trie remaining the better choice for large datasets thanks to its efficient memory structure and binary search behavior.

Part d: Compressed Trie with Ranking Algorithm Analysis

To further optimize our text auto completion system, we considered implementing a compressed trie (also known as a radix or Patricia trie). Unlike our current standard trie, which stores every character individually even if a node has only one child. A compressed trie merges chains of nodes with single children into a single node. This significantly reduces redundancy, especially when many words share common prefixes.

Considering our current implementation, if we integrate a compressed trie, the time complexity for insertion, search, and autocomplete operations remains O(L), where L is the length of the word or prefix. Although the compressed trie reduces the number of nodes by merging linear paths, each operation still requires character-by-character comparisons along the stored substrings. This matches closely with our existing standard trie implementation.

In terms of space complexity, the compressed trie offers considerable efficiency, bringing it down to O(N), where N is the total number of words. This occurs because redundant paths common to multiple words are consolidated into fewer nodes. As we learned in lecture, the trie structure exploits common prefixes to minimize storage, and the compressed trie maximizes this by reducing repeated nodes into single representative nodes. The ranking algorithm we analyzed previously, which tracks word frequency to influence search result order, slightly increases space complexity by O(N) to store frequency counters but does not significantly impact the overall complexity of the trie structure itself.

Thus, the compressed trie with a ranking algorithm remains highly space-efficient (O(N)) and maintains efficient operation times (O(L)), providing an ideal solution for handling large datasets due to significant memory optimization.

Part e: Deploying Standard Trie Using an Array

We also explored whether our current trie implementation could be modified to use arrays instead of dictionaries for child nodes. In this scenario, each trie node would contain an array of size 26, corresponding to each lowercase English letter. This approach allows constant-time access (O(1)) for child nodes by direct indexing using ASCII values, maintaining the same insertion, search, and autocomplete time complexity of O(L), similar to our dictionary-based trie.

However, the trade-off appears significantly in terms of space complexity. The space complexity becomes O(26 × N × L), where N is the number of words, and L is their average length. Every node, regardless of whether it has fewer than 26 children, allocates space for all possible characters. This results in a substantial waste of memory compared to our dictionary implementation, which dynamically adjusts to store only existing children.

Therefore, while the array implementation provides rapid access speed, its excessive memory usage makes it inefficient for applications where space is a significant constraint.

Part f: Deploying Standard Trie Using a LinkedList

Finally, we assessed the viability of using a linked list to manage children nodes in a standard trie. Although possible, implementing trie nodes with linked lists significantly impacts operational efficiency.

With linked lists, insertion and search operations require traversing through potentially all 26 child nodes at each level to find the correct character. This traversal process drastically increases the time complexity to O(26L), simplifying to O(L), yet practically being slower due to constant factors in real-world scenarios. Each level’s linear traversal introduces inefficiency compared to the constant-time dictionary or array implementations.

Regarding space complexity, the linked list implementation would yield $O(N \times L)$, similar to our existing dictionary-based standard trie. Despite avoiding the fixed-size overhead of arrays, linked lists introduce additional overhead due to pointer storage and node management.

In conclusion, although the linked list method does not worsen the theoretical space complexity substantially, its practical time complexity and overall performance degradation render it unsuitable for our autocompletion system, particularly when considering larger datasets or the need for rapid autocomplete response times.